

Test Driven Cookbook Development



Welcome to Test Driven Cookbook Development.

Slide 2

Franklin Webber

Training and Technical Content Lead

US Army - Radio/COMSEC Repairer (35E)

Tech Support

Quality Assurance

Software Developer Eng. in Test

Software Developer

Instructor / Trainer



Before we start let me introduce myself.

Slide 3

You

Chef Cookbook Author

Great Looking

Understand Chef Resources and Cookbooks

Interested in Testing and know a little about Chef Testing Tools



©2016 Chef Software Inc.

1-3




Let me tell who I think you are and for whom this content has been designed.

Slide 4

Schedule

- 3 min – Introductions & Housekeeping
- 39 min – Slides & Live Demonstration
- 15 min – Questions
- 3 min – Wrap-up

©2016 Chef Software Inc. 1-4



This is the general schedule of the presentation.

Slide 5



Welcome again and let's get started.

Slide 6

Agenda

- ❑ What is Test Driven Development (TDD)?
- ❑ Writing a test first
- ❑ Refactoring with tests
- ❑ Faster Feedback with Unit Tests
- ❑ Further Resources


We are going to cover a lot of content in the next 39 minutes. We will first start with defining what is Test Driven Development.

Slide 7


CONCEPT

Test Driven Development

1. Define a test set for the unit first
2. Then implement the unit
3. Finally verify that the implementation of the unit makes the tests succeed.
4. Refactor



©2016 Chef Software Inc. 1-7




Test Driven Development (TDD) is a workflow that asks you to perform that act continually and repeatedly as you satisfy the requirements of the work you have chosen to perform.

TDD generically focuses on the unit of software any level. It is the process of writing the test first, implementing the unit, and then verifying the implementation with the test that was written.

A 'unit' of software is purposefully vague. This 'unit' is definable by the individuals developing the software. So the size of a 'unit of software' likely has different meanings to different individuals based on our backgrounds and experiences.

Slide 8

CONCEPT




Behavior Driven Development (BDD)

Behavior-driven development (BDD) specifies that tests of any unit of software should be specified in terms of the desired behavior of the unit.

Borrowing from [agile software development](#) the "desired behavior" in this case consists of the requirements set by the business — that is, the desired behavior that has [business value](#) for whatever entity commissioned the software unit under construction.

Within BDD practice, this is referred to as BDD being an "outside-in" activity.

©2016 Chef Software Inc. 1-8 

How you choose to express the requirements of that unit is the crux of Behavior Driven Development (BDD). Behavior-driven development specifies that tests of any unit of software should be specified in terms of the desired behavior of the unit. Expressing this desired behavior is often expressed in scenarios that are written in a Domain Specific Language (DSL).

The cookbooks and recipes that you have written so far share quite a few similarities with BDD. In Chef, you express the desired state of the system through a DSL, resources, you define in recipes.


Slide 9


CONCEPT

TDD and BDD

TDD is a workflow process.

BDD influences the language we use to write tests and how we focus on the tests that matter.



©2016 Chef Software Inc. 1-9 

TDD is a workflow process: Add a test; Run the test expecting failure; Add code; Run the test expecting success. Refactor.

BDD influences the language we use to write the tests and how we focus on test that matter. The activities within this module focus on the process of taking requirements, expressing them as expectations, choosing one implementation to meet these expectations, and then verifying we have met these expectations.

Building a Web Server

1. Install the httpd package
2. Write out a test page
3. Start and enable the httpd service

To explore the concepts of Test Driven Development through Behavior Driven Design we are going to focus on creating a cookbook that starts with the goal that installs, configures, and starts a web server that hosts the your company's future home page. The goal again is to focus on the TDD workflow and understanding how to apply BDD when defining these tests. We are not concerned about focusing on best practices for managing web servers or modeling a more initially complex cookbook.

Scenario: Potential User Visits Website

Given that **I am a potential user**

When **I visit the company website in my browser**

Then I should **see a welcome message**

The typical reason for setting up a website is to allow customers, users, potential users to learn more about the company. The needs of the website may change in the future but the first minimum viable product (MVP) is to simply give our users the ability to find out more information.

Our goal now is to define a scenario with this understanding.

This first scenario is enough information to help us build this cookbook with a TDD approach. This practice of defining a scenario is a tactic that I employ to help focus me on the most valuable work that needs to be done.

Important things to notice in the following scenario is the distinct lack of technology or implementation. The scenario is not concerned about the services that are running or files that might be found on the file system.


Agenda

- ✓ What is Test Driven Development (TDD)?
- ❑ Writing a test first
- ❑ Refactoring with tests
- ❑ Faster Feedback with Unit Tests
- ❑ Further Resources


Now that we have laid the foundation and described the workflow. It is now time for me to show you how it is done.

Slide 13

Let's Start this Journey in the Home Directory



A terminal window icon is shown on the left. The terminal window has a black background with white text. The prompt is `>` and the command entered is `cd ~`. The rest of the terminal window is empty.

©2016 Chef Software Inc. 1-13 

Let's start the journey on your workstation. From the home directory we are going to creating this cookbook.

Generate a Cookbook



```
> chef generate cookbook httpd
```

```
Compiling Cookbooks...
Recipe: code_generator::cookbook
  * directory[/home/chef/httpd] action create
    - create new directory /home/chef/httpd
  * template[/home/chef/httpd/metadata.rb] action
create_if_missing
    - create new file /home/chef/httpd/metadata.rb
    - update content in file /home/chef/httpd/metadata.rb from
none to 53a150
    (diff output suppressed by config)
  * template[/home/chef/httpd/README.md] action create_if_missing
```

Let's generate cookbook named 'httpd'. The name of the cookbook here resembles the name of a public cookbook in the Supermarket that accomplishes a very similar task. That is the reason why I have asked you to choose the same name.

Sharing the same name as a cookbook within the Supermarket can be problematic. While we may never share this cookbook other individuals within our organization could believe it to be a copy of that cookbook. When it comes to naming cookbooks it may be wise to first search the Supermarket and ensure you are not using a similar name.

View the Tests in the Generated Cookbook



```
> tree httpd
```

```
httpd
├── test
│   ├── integration
│   │   ├── default
│   │   │   └── serverspec
│   │   │       └── default_spec.rb
│   └── helpers
│       ├── serverspec
│       └── spec_helper.rb
```

We can examine the contents of the cookbook that chef generated for us. Here you see that the tool created for us a complete test directory structure.

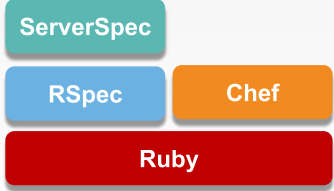

Slide 16

CONCEPT

RSpec and ServerSpec


RSpec is a Domain Specific Language (DSL) that allows you to express and execute expectations. These expectations are expressed in examples that are asserted in different example groups.

ServerSpec provides helpers and tools that allow you to express expectations about the state of infrastructure.



©2016 Chef Software Inc.

1-16



RSpec is a Behavior Driven Development (BDD) framework that uses a natural language domain-specific language (DSL) to quickly describe scenarios in which systems are being tested. RSpec allows you to setup a scenario, execute the scenario, and then define expectations on the results. These expectations are expressed in examples that are asserted in different example groups.

RSpec by itself grants us the framework, language, and tools. ServerSpec provides the knowledge about expressing expectations about the state of infrastructure.

Auto-generated Spec File in Cookbook

 ~/httpd/test/integration/default/serverspec/default_spec.rb


```
require 'spec_helper'

describe 'httpd::default' do
  # Serverspec examples can be found at
  # http://serverspec.org/resource_types.html
  it 'does something' do
    skip 'Replace this with meaningful tests'
  end
end
```

The generator created an example specification (or spec) file. Before we talk about the RSpec/ServerSpec language lets explain the long file path and its importance.

Slide 18

CONCEPT



Where do Tests Live?


```
~/httpd/test/integration/default/serverspec/default_spec.rb
```

Test Kitchen will look for tests to run under this directory. It allows you to put unit or other tests in test/unit, spec, acceptance, or wherever without mixing them up. This is configurable, if desired.

<http://kitchen.ci/docs/getting-started/writing-test>

©2016 Chef Software Inc.


1-18



Let's take a moment to describe the reason behind this long directory path. Within our cookbook we define a test directory and within that test directory we define another directory named 'integration'. This is the basic file path that Test Kitchen expects to find the specifications that we have defined.

Slide 19

CONCEPT



Where do Tests Live?


```
~/httpd/test/integration/default/serverspec/default_spec.rb
```

This corresponds exactly to the Suite name we set up in the .kitchen.yml file. If we had a suite called "server-only", then you would put tests for the server only suite under.

<http://kitchen.ci/docs/getting-started/writing-test>

©2016 Chef Software Inc.


1-19


CHEF

The next part the path, 'default', corresponds to the name of the test suite that is defined in the .kitchen.yml file. In our case the name of the suite is 'default' so when test kitchen performs a `kitchen verify` for the default suite it will look within the 'default' folder for the specifications to run.

Slide 20

CONCEPT



Where do Tests Live?


```
~/httpd/test/integration/default/serverspec/default_spec.rb
```

This tells Test Kitchen (and Busser) which Busser runner plugin needs to be installed on the remote instance.

<http://kitchen.ci/docs/getting-started/writing-test>

©2016 Chef Software Inc.


1-20



'serverspec' is the kind of tests that we want to define. Test Kitchen supports a number of testing frameworks.

Slide 21

CONCEPT




Where do Tests Live?

```
~/httpd/test/integration/default/serverspec/default_spec.rb
```

All test files (or specs) are named after the recipe they test and end with the suffix "_spec.rb". A spec missing that will not be found when executing `kitchen verify`.

<http://kitchen.ci/docs/getting-started/writing-test>

©2016 Chef Software Inc. 1-21 

'serverspec' is the kind of tests that we want to define. Test Kitchen supports a number of testing frameworks.

Now that we understand the path let's take a look at the RSpec/ServerSpec language.

Slide 22

ServerSpec Example

```
describe 'httpd::default' do
  describe command('curl http://localhost') do
    its(:stdout) { should match(/Welcome Home/) }
  end
end
```

A red bracket on the right side of the code block groups the two nested `describe` blocks. A red rectangular box points to this bracket with the text "example groups".

example groups

ServerSpec allows you to scope the expectations you write into groups. These groups are given names or ServerSpec resources to help describe their scope. Example groups can be nested and often are to describe a hierarchical structure of state.

In this example we see an outer example group with the name 'httpd::default' and the inner example group with the ServerSpec resource named `command`. This means that the `command` example group is within the example group 'httpd::default'.

Example groups can be used to show relationship, like they are done here. They can also be used to create unique scenarios with different states for the system.

Slide 23

Components of a ServerSpec Example

```
describe 'httpd::default' do
  describe command('curl http://localhost') do
    its(:stdout) { should match(/Welcome Home/) }
  end
end
```

Diagram labels and connections:

- cookbook name::suite name** points to `'httpd::default'`
- ServerSpec resource** points to `command('curl http://localhost')`
- attribute of resource** points to `its(:stdout)`
- expectation** points to `should match(/Welcome Home/)`

http://serverspec.org/resource_types.html

The outermost describe here represents the cookbook name and the suite name. The inner expectation is using a ServerSpec resource named command. This command resource takes a parameter which is the command it will run. Within the inner example group we are asking for an attribute on the ServerSpec command resource. A command resource is one of the few resources that has attributes you can query. In this instance the example is asking for the standard out, abbreviated as 'stdout'. Lastly we express an expectation that we expect the actual result returned from the standard out to contain the text 'Welcome Home'.

Slide 24

Remove the Default Test

~/httpd/test/integration/default/serverspec/default_spec.rb

```
require 'spec_helper'

describe 'httpd::default' do
  # Serverspec examples can be found at
  # http://serverspec.org/resource_types.html
  it 'does something' do
    skip 'Replace this with meaningful tests'
  end
end
```

Within the test file found at the following path you will find that it is already populated some initial code. The first two lines are comments to the ServerSpec documentation. The next three lines are a placeholder test that when executed notifies you that this test is skipped.

We do not need these comments or the placeholder test so let's remove it from the test file.

Add a Test to Validate a Working Website

~/httpd/test/integration/default/serverspec/default_spec.rb

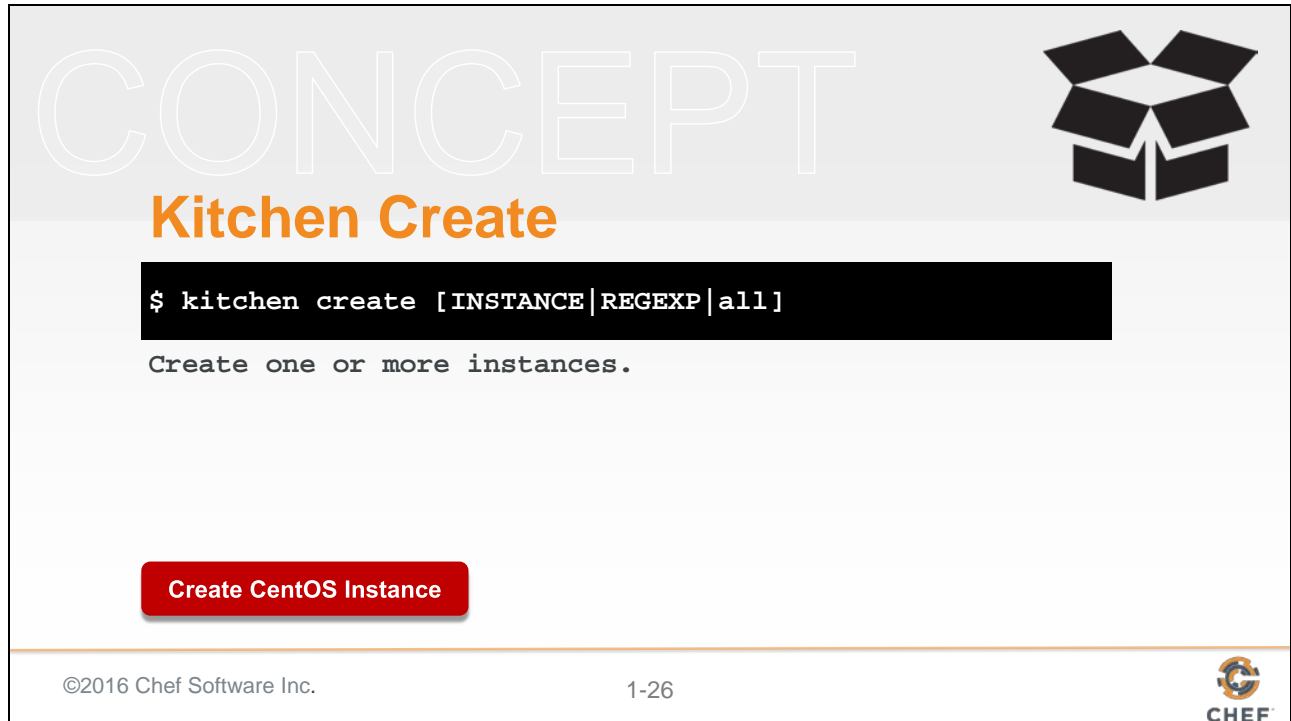
```
require 'spec_helper'

describe 'httpd::default' do
  describe command('curl http://localhost') do
    its(:stdout) { should match(/Welcome Home/) }
  end
end
```

Define the following expectation that states that when we would visit that site it should return a welcoming message.

ServerSpec provides a helper method that allows you to specify a command. That command returns the results from the command through standard out. We are asking the command's standard out if anywhere in the results match the value 'Welcome Home'.

Slide 26



The slide features a light gray background with the word "CONCEPT" in large, white, outlined letters at the top left. To the right is a black icon of an open box. Below "CONCEPT" is the title "Kitchen Create" in bold orange text. A black rectangular box contains the command `$ kitchen create [INSTANCE|REGEXP|all]` in white. Below this box, the text "Create one or more instances." is displayed in a small, gray, monospaced font. At the bottom left, there is a red button with the text "Create CentOS Instance" in white. The footer consists of a thin orange line, followed by "©2016 Chef Software Inc." on the left, "1-26" in the center, and the Chef logo on the right.

CONCEPT

Kitchen Create

```
$ kitchen create [INSTANCE|REGEXP|all]
```

Create one or more instances.

Create CentOS Instance

©2016 Chef Software Inc. 1-26 CHEF

Create or turn on a virtual or cloud instance for the platforms specified in the kitchen configuration.

Running 'kitchen create default-centos-67' would create the the one instance that uses the test suite on the platform we want.


Typing in that name would be tiring if you had a lot of instances. A shortcut can be used to target the same system 'kitchen create default' or 'kitchen create centos' or even 'kitchen create 67'. This is an example of using the Regular Expression (REGEXP) to specify an instance.

When you want to target all of the instances you can run 'kitchen create' without any parameters. This will create all instances. Seeing as how there is only one instance this will work well.

In our case, this command would use the Docker driver to create a docker image based on centos-6.7.

Slide 27

CONCEPT



Kitchen Converge

```
$ kitchen converge [INSTANCE|REGEXP|all]
```

Create the instance (if necessary) and then apply the run list to one or more instances.


Create CentOS Instance

→

Install Chef

→

Apply the Run List

©2016 Chef Software Inc. 1-27 

Creating an image gives us a instance to test our cookbooks but it still would leave us with the work of installing chef and applying the cookbook defined in our `.kitchen.yml` run list.


So let's introduce you to the second kitchen command: 'kitchen converge'.

Converging an instance will create the instance if it has not already been created. Then it will install chef and apply that cookbook to that instance.

In our case, this command would take our image and install chef and apply the `httpd` cookbook's default recipe.

Slide 28


CONCEPT




Kitchen Verify

```
$ kitchen verify [INSTANCE|REGEXP|all]
```

Create, converge, and verify one or more instances.



```
graph LR; A[Create CentOS Instance] --> B[Install Chef]; B --> C[Apply the Run List]; C --> D[Execute Tests]
```

©2016 Chef Software Inc. 1-28 

To verify an instance means to:

- Create a virtual or cloud instances, if needed
- Converge the instance, if needed
- And then execute a collection of defined tests against the instance

Create the Virtual Instance



```
> kitchen create
```

```
-----> Starting Kitchen (v1.4.2)
-----> Creating <default-centos-67>...
    Sending build context to Docker daemon 26.11 kB
    Sending build context to Docker daemon
    Step 0 : FROM centos:centos6
    centos6: Pulling from centos
    47d44cb6f252: Pulling fs layer
    ...
    Finished creating <default-centos-67> (2m28.65s).
-----> Kitchen is finished. (2m29.39s)
```

Create the instance with the following command. Here Test Kitchen will ask the driver specified in the kitchen configuration file to provision an instance for us.

Converge the Virtual Instance



```
> kitchen converge
```

```
----> Starting Kitchen (v1.4.2)
-----> Converging <default-centos-67>...
$$$$$$ Running legacy converge for 'Docker' Driver
...
-----> Installing Chef Omnibus (install only if missing)
Downloading https://www.chef.io/chef/install.sh to file...
resolving cookbooks for run list: ["httpd::default"]
...
Finished converging <default-centos-67> (0m27.64s).
-----> Kitchen is finished. (0m28.58s)
```

Creating the instance allows us to view the operating system but Chef is not installed and the cookbook recipe, defined in the run list of the default test suite, has not been applied to the system. To do that you need to run 'kitchen converge'. Converge will take care of all of that.

In this instance the default recipe of the httpd cookbook contains no resources. You have not written a single resource that defines your desired state. Before we do that we want to ensure the instance is not already in a state that perhaps already meets the expectations that we defined.

Execute the Tests Against the Virtual Instance



```
> kitchen verify
```

```
-----> Starting Kitchen (v1.4.2)
-----> Setting up <default-centos-67>...
-----> Installing Busser (busser)
-----> Verifying <default-centos-67>...
-----> Running serverspec test suite
-----> Installing Serverspec..
-----> serverspec installed (version 2.24.1)
      /opt/chef/embedded/bin/ruby -
I/tmp/verifier/suites/serverspec -I/tmp/verifier/gems/gems/rspec-
support-3.3.0/lib:/tmp/verifier/gems/gems/rspec-core-3.3.2/lib
/opt/chef/embedded/bin/rspec --pattern
```

To verify the state of the instance with specification that we defined we use the 'kitchen verify' command. This command will install all the necessary testing tools, configure them, and then execute the test suite, and return to us the results.

Something that is important to mention is that we could have simply run this command from the start. When no previous instance exists, no instance has been created or converged, this command will automatically perform those two steps. When the instance is running, however, the verification step is only run.

Slide 32

Understanding the Failure Message

```
>>>>> Verify failed on instance <default-centos-67>.
>>>>> Please see .kitchen/logs/default-centos-67.log for more details
>>>>> -----Exception-----
>>>>> Class: Kitchen::ActionFailed
>>>>> Message: SSH exited (1) for command: [sh -c '

BUSSEER_ROOT="/tmp/verifier"; export BUSSEER_ROOT
GEM_HOME="/tmp/verifier/gems"; export GEM_HOME
GEM_PATH="/tmp/verifier/gems"; export GEM_PATH
GEM_CACHE="/tmp/verifier/gems/cache"; export GEM_CACHE

sudo -E /tmp/verifier/bin/busser test
']
>>>>> -----
[chef@ip-172-31-14-170 httpd]$
```

kitchen failure

command executed

Now, let's read the results from the kitchen verification to ensure that our expectations failed to be met.

When the command completes you will see a block of code that tells you that the verification failed and that an exception has occurred. Immediately your eyes will start to scan this block of text for some information about the failure and unfortunately you will not see anything to help you understand what is happening. Because what you are looking at is the test command executed by the Busser on the test instance but not the results of the command. To see those results you will need to scroll back up in your history.

Examine the Test Kitchen Results

```
-----> serverspec installed (version 2.24.1)
/opt/chef/embedded/bin/ruby -I/tmp/verifier/suites/serverspec
-I/tmp/verifier/gems/gems/rspec-support-3.3.0/lib:/tmp/.../rspec-
core-3.3.2/lib /opt/chef/embedded/bin/rspec --pattern
/tmp/verifier/suites/serverspec/*/*/*_spec.rb --color --format
documentation --default-path /tmp/verifier/suites/serverspec
```

ServerSpec is the default verifier for Test Kitchen. The Busser tool installs it, configures it, and the executes it for you on the test instance.

Scroll back until you can find the message that tells you that serverspec is installed.

Examine the Test Kitchen Results

```
-----> serverspec installed (version 2.24.1)
/opt/chef/embedded/bin/ruby -I/tmp/verifier/suites/serverspec
-I/tmp/verifier/gems/gems/rspec-support-3.3.0/lib:/tmp/.../rspec-
core-3.3.2/lib /opt/chef/embedded/bin/rspec --pattern
/tmp/verifier/suites/serverspec/*/*/*_spec.rb --color --format
documentation --default-path /tmp/verifier/suites/serverspec
```

ruby executable

include ruby libraries

rspec executable

rspec parameters

The line that immediately follows it is the command that is executed on the system. The ServerSpec verifier is running ruby, loading up the test suite libraries, executing the command rspec. The rspec command is being provided a number of command-line parameters that tell it: where to find the test files and what they look like; to colorize the output; and how to output the results.

Slide 35

Examine the RSpec Results

```
httpd::default
  Command "curl http://localhost"
    stdout
      should match /Welcome Home/ (FAILED -1)
```

Failure Number

RSpec displays a summary of the results in the 'documentation' format. This format allows us to read the example groups and see that:

When running the specified command the standard out failed to match the value 'Welcome Home' anywhere within the results.

The results are displayed in color and in RSpec's documentation format. This shows us the example that we wrote in a hierarchal view. The indentation is intentional to show the nested relationships of the example groups and the example. The expectation in the example you defined failed, as we expected. The text will be displayed in red and provide a failure number. Details about the failure will be displayed below.

Slide 36

Examine Failure #1

```
1) httpd::default Command "curl http://localhost" stdout should match ...  
Failure/Error: its(:stdout) { should match(/Welcome Home/) }  
  expected "" to match /Welcome Home/  
  Diff:  
  @@ -1,2 +1,2 @@  
  -/Welcome Home/  
  +"  
  /bin/sh -c curl\ http://localhost  
# /.../serverspec/default_spec.rb:7:in `block (3 levels) in ...
```

actual results

difference

system command

spec file : line number

Each failure is displayed with their failure number, in order, in more detail in a failures section. A failure contains a number of details about the failure.

First it will display a sentence created out of the example groups and example that we defined. Below that it will display all the details about the failure that include: the actual results that were received; the difference between the actual and the expected results; the command run against the virtual instance; and the spec file and line number within that spec file where the failing expectation can be found.

Slide 37

Examine the Test Summary

```
Finished in 0.20256 seconds (files took 0.60564 seconds to load)  
1 example, 1 failure
```

A final summary contains the length of execution time with the results shows that RSpec verified 1 example and found 1 failure.

After all the failures a final summary of the results will be displayed which shows us that our test suite contains 1 example and that 1 example failed to meet expectations.

Slide 38

Write the Default Recipe for the Cookbook

```
~/httpd/recipes/default.rb

#
# Cookbook Name:: httpd
# Recipe:: default
#
# Copyright (c) 2016 The Authors, All Rights Reserved.
package 'httpd'

file '/var/www/html/index.html' do
  content '<h1>Welcome Home!</h1>'
end

service 'httpd' do
  action [:enable, :start]
end
```

The following recipe defines three resources. These three resources express the desired state of an apache server that will serve up a simple page that contains the text 'Welcome Home!'.

The package will install all the necessary software on the operating system. The file will create an HTML file with the desired content at a location pre-defined by the web server. The service resource will start the web server and then ensure that if we reboot the system the web server will start up.

Re-Converge the Virtual Instance



```
> kitchen converge
```

```
-----> Starting Kitchen (v1.4.2)
Converging 2 resources
Recipe: httpd::default
  * package[httpd] action install
    - install version 2.2.15-47.el6.centos of package httpd
  * file[/var/www/html/index.html] action create
    - ...
  * service[httpd] action enable
    - enable service service[httpd]
  * service[httpd] action start
    - start service service[httpd]
```

Whenever you make a change to the recipe it is important to run 'kitchen converge'. This command will apply the updated recipe to the state of the virtual instance.

In the output, you should see the resources that you defined being applied to the instance. The package, the file, and the actions of the service.

Slide 40

Re-Verify the Virtual Instance



```
> kitchen verify
```

```
httpd::default
```

```
  Command "curl http://localhost"
```

```
    stdout
```

```
      should match /Welcome Home/
```

To verify the state of the virtual instance you run the 'kitchen verify' command. In the summary you should find the failing expectation no longer fails.

If it does, it is time to review the code you wrote in the recipe file and the spec file. When it was failing did you get a different failure than the one that we walked through? That probably means there is an error in the spec file. Did the test instance actually converge successfully? Sometimes output will scroll by and we don't have time to read it. I get it. Scroll back up and see if there was an error message tucked into the 'kitchen converge' you ran.

Agenda

- ✓ What is Test Driven Development (TDD)?
- ✓ Writing a test first
- ❑ Refactoring with tests
- ❑ Faster Feedback with Unit Tests
- ❑ Further Resources

Now with the first test described it is time to begin the final step of the TDD. Refactoring.

Slide 42

CONCEPT

Test Driven Development

1. Define a test set for the unit first
2. Then implement the unit
3. Finally verify that the implementation of the unit makes the tests succeed.
4. **Refactor**

Did someone say refactor?

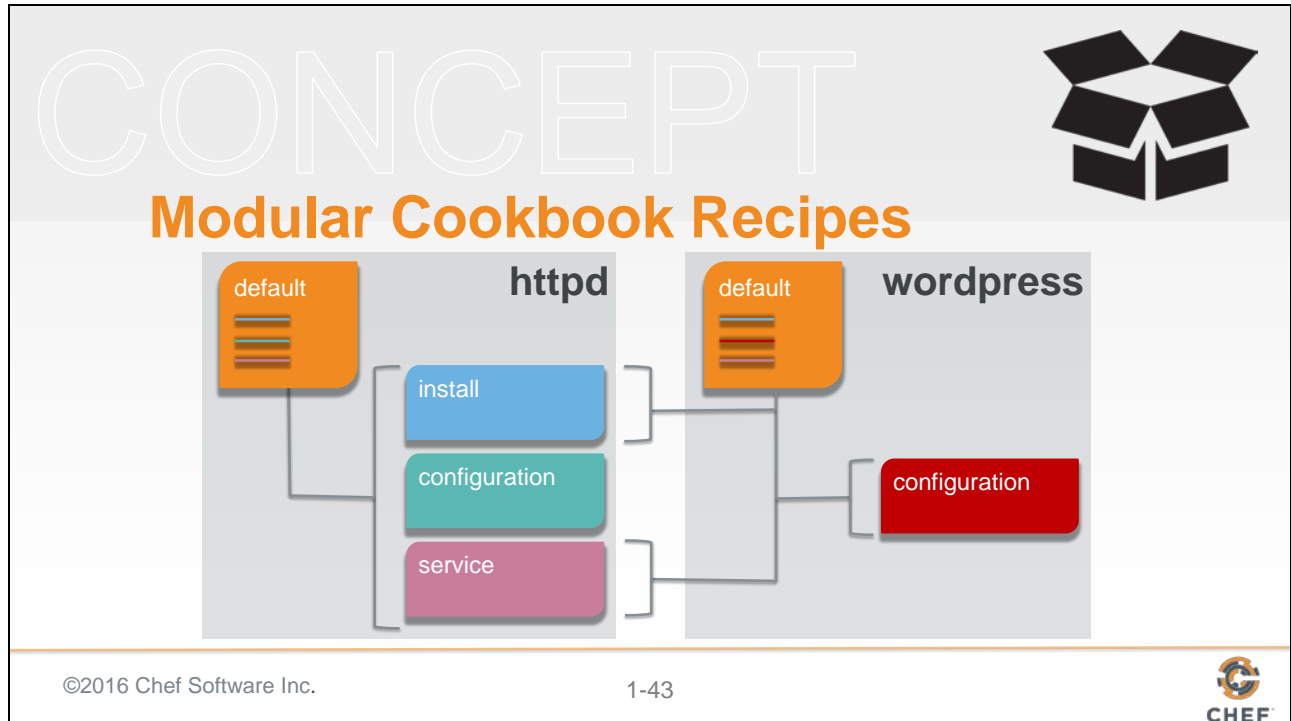


©2016 Chef Software Inc. 1-42

Refactoring is the often forgotten step in the TDD cycle. When we are able to get our expectations to pass we immediately want to move to our next requirement or next cookbook.

This step is incredibly important. Within it we are able to reflect on the unit of code and tests that we have written and evaluate them. How you evaluate the code may vary based on your experience, the standards defined by the team you work with, or if the code will be shared with the Chef community.

Slide 43




Our initial implementation of the default recipe for the httpd cookbook defined the entire installation, configuration, and management of the service within a single recipe. This implementation has the benefit of being entirely readable from a single recipe. However, it does not easily allow for other cookbooks that may want to use the httpd cookbook to easily choose the components that it may need.

An example of this is that we may deploy wordpress or some other web application that relies on the apache webserver installed and running. In this new cookbook we would like to re-use the resources of the content that installs apache and the resources that manage the service. We most likely do not want to setup a test page that greets people. We are likely going to replace it with application code.

Slide 44


CONCEPT



include_recipe

A recipe can include one (or more) recipes located in cookbooks by using the `include_recipe` method. When a recipe is included, the resources found in that recipe will be inserted (in the same exact order) at the point where the `include_recipe` keyword is located.

<https://docs.chef.io/recipes.html#include-recipes>

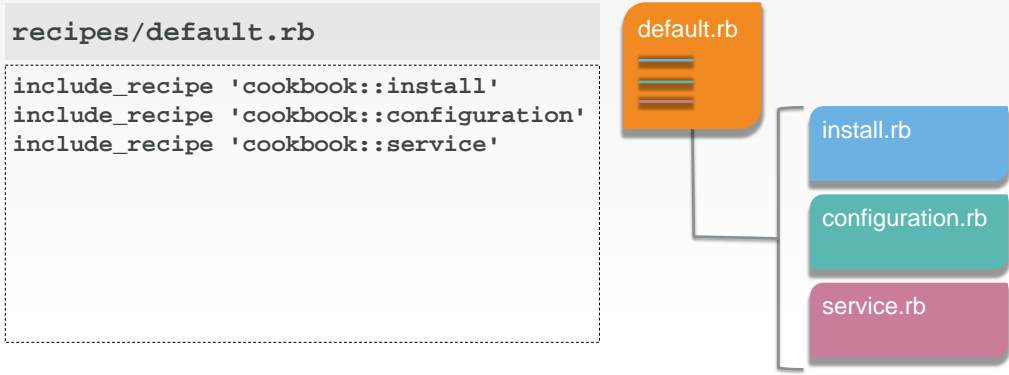
©2016 Chef Software Inc. 1-44 

The 'include_recipe' method can be used to include recipes from the same cookbook or external cookbooks. It allows us to accomplish what we saw previously. This gives us the ability to build recipes in more modular ways promoting better re-use patterns within the cookbooks we write.

Slide 45

CONCEPT

Recipe Organization




```
recipes/default.rb
```

```
include_recipe 'cookbook::install'
include_recipe 'cookbook::configuration'
include_recipe 'cookbook::service'
```

default.rb

- install.rb
- configuration.rb
- service.rb

©2016 Chef Software Inc. 1-45



To allow better re-use we can choose to refactor a single recipe into more modular recipes that focus on their individual concerns. Then these recipes can be included into the original single recipe through the 'include_recipe' method.

Slide 46

EXERCISE



Refactor to Modular Recipes

This is why we can have nice things!

Objective:

- ☐ Refactor the installation into a separate recipe
- ☐ Converge the cookbook and execute the tests

Would you please take a little off the top.



©2016 Chef Software Inc. 1-46

This more modular approach to recipes is very common as the complexity of the cookbook continues to grow. The complexity of the cookbook we are developing is not there, nor will it ever be there for the entirety of this course. However, we are still going to use this opportunity to prematurely optimize to demonstrate the refactoring of a cookbook.

Together we will work through creating a recipe that manages the installation of the webserver.

Generate an Install Recipe



```
> chef generate recipe install
```

```
Recipe: code_generator::recipe
  * directory[/home/chef/httpd/spec/unit/recipes] action create
  (up to date)
  * cookbook_file[/home/chef/httpd/spec/spec_helper.rb] action
  create_if_missing (up to date)
  * template[/home/chef/httpd/spec/unit/recipes/install_spec.rb]
  action create_if_missing
    - create new file
    /home/chef/httpd/spec/unit/recipes/install_spec.rb
    - update content in file
    /home/chef/httpd/spec/unit/recipes/install_spec.rb from none to
    187413
```

Since we are within the cookbook directory you simply need to provide it the name of the recipe you want created.

Slide 48

Write the Install Recipe

```
~/httpd/recipes/install.rb

#
# Cookbook Name:: httpd
# Recipe:: install
#
# Copyright (c) 2016 The Authors, All Rights Reserved.
package 'httpd'
```

The installation of the web server can be expressed with this one resource. Within the new recipe add the following resource.

Slide 49

Remove the Resource from the Default Recipe

 `~/httpd/recipes/default.rb`

```
#  
# Cookbook Name:: httpd  
# Recipe:: default  
#  
# Copyright (c) 2016 The Authors, All Rights Reserved.  
package 'httpd'  
  
file '/var/www/html/index.html' do  
  content '<h1>Welcome Home!</h1>'  
end  
  
service 'httpd' do  
  action [:enable, :start]  
end
```

Now that we have defined the installation of the webserver in a separate recipe it is time to remove the installation from the default recipe.

Include the Install Recipe

 ~/httpd/recipes/default.rb

```
#
# Cookbook Name:: httpd
# Recipe:: default
#
# Copyright (c) 2016 The Authors, All Rights Reserved.
include_recipe 'httpd::install'


file '/var/www/html/index.html' do
  content '<h1>Welcome Home!</h1>'
end

service 'httpd' do
  action [:enable, :start]
end
```

Replacing it with the 'include_recipe' method that retrieves the contents of that recipe and includes it here.

Slide 51

EXERCISE



Refactor to Modular Recipes

This is why we can have nice things!

Objective:

- ✓ Refactor the installation into a separate recipe
- ❑ Converge the cookbook and execute the tests

Now to see how I look in the mirror



©2016 Chef Software Inc. 1-51

The default recipe has changed. It is now time to ensure that we did everything right by converging the latest changes against the test instance and then verifying the changes by executing our tests.

Re-Converge the Test Instance



```
> kitchen converge
```

```
----> Starting Kitchen (v1.4.2)
----> Converging <default-centos-67>...
$$$$$$ Running legacy converge for 'Docker' Driver
...
----> Installing Chef Omnibus (install only if missing)
      Downloading https://www.chef.io/chef/install.sh to file...
      resolving cookbooks for run list: ["httpd::default"]
      ...
      Finished converging <default-centos-67> (0m27.64s).
----> Kitchen is finished. (0m28.58s)
```

Whenever a change is made to a recipe or component of the cookbook it is important to converge the latest cookbook against the test instance.

If an error occurs that likely means that you have a typo within your default recipe or the install recipe.

Re-Verify the Test Instance




```
> kitchen verify
```

```
httpd::default
  Command "curl http://localhost"
  stdout
    should match /Welcome Home/
```

If everything converges successfully it is time to verify the state of the instance with the test that we have defined.

Slide 54

EXERCISE




Refactor to Modular Recipes

This is why we can have nice things!

Objective:

- ✓ Refactor the installation into a separate recipe
- ✓ Converge the cookbook and execute the tests

Yes, I would like some product in my hair.




©2016 Chef Software Inc. 1-54

Together we were able to refactor the cookbook while implementing the installation recipe.


Slide 55

DISCUSSION



Do Our Tests Really Work?

What if we removed code from within the recipes and ran the tests?

©2016 Chef Software Inc. 1-55 

During the group exercise and the lab we made changes to the recipes that we were able to verify on the test instance. If you accidentally or purposefully created a typo for yourself you would have seen the converge or the verification fail. However, what if removed code from the recipes that we wrote?


The omission (or in this case removal of code) of resources could have happened. When we refactored the default recipe we may have remembered to remove the resources that manage the configuration but forgot to use the 'include_recipe' to ensure we loaded the new recipe. Or it is possible that we created a service recipe that we never populated but made all the appropriate changes to the default recipe.


Slide 56

CONCEPT

Heckling Your Code

Mutation testing is used to design new software tests and evaluate the quality of existing software tests. Mutation testing involves modifying a program in small ways.





©2016 Chef Software Inc. 1-56 

Removing code sabotages the policy that you have defined. If you used Test Kitchen to converge and verify the cookbook and saw a failure you can sleep soundly at night knowing your tools have you covered. On the other hand, if Test Kitchen were to return success, after such a change, then it might cause you to break out in a cold sweat.

Removing code from a recipe or recipes is a small change. So is introducing a typo into the code, specifying a different resource name or changing the value of a resource attribute. The process of modifying the code in small ways and then executing the test suite against it is often times referred to as mutation testing.

Slide 57

EXERCISE




Heckle That Code

It could be a game show. Maybe on Twitch?

Objective:

- ☐ Remove / Comment source code
- ☐ Converge the cookbook and execute the tests

©2016 Chef Software Inc. 1-57 

Before we leave this module, let's do a little mutation testing, to ensure the test that we have defined is good enough.

Slide 58

Comment Out Key Code Within the Default Recipe

 `~/httpd/recipes/default.rb`

```
#  
# Cookbook Name:: httpd  
# Recipe:: default  
#  
# Copyright (c) 2016 The Authors, All Rights Reserved.  
# include_recipe 'httpd::install'  
  
file '/var/www/html/index.html' do  
  content '<h1>Welcome Home!</h1>'  
end  
  
service 'httpd' do  
  action [:enable, :start]  
end
```

Return to the default recipe and choose one line to remove or comment out. Here I have chosen to comment out the first line that includes the install recipe.

Re-Converge the Test Instance



```
> kitchen converge
```

```
-----> Converging <default-centos-67>...  
Synchronizing Cookbooks:  
  - httpd (0.1.0)  
Compiling Cookbooks...  
Converging 2 resources  
Recipe: httpd::configuration  
  (up to date)  
Recipe: httpd::service  
  (up to date)  
    * service[httpd] action enable (up to date)
```

When converging the updated recipe it no longer shows the install recipe being loaded. This has changed the number of resources that are converged on the test instance. Removing the recipe from the default recipe does not remove any of the components that it previously installed.

Re-Verify the Test Instance




```
> kitchen verify
```

```
httpd::default
  Command "curl http://localhost"
  stdout
    should match /Welcome Home/
Finished in 0.15802 seconds (files took 0.63276 seconds ...)
1 example, 0 failures
```

Verification of the test instance will return a success. Despite removing the install recipe from the default recipe the test instance is still able to serving the default web page that our test is looking for when it requests data from the site.

Slide 61


CONCEPT



Kitchen Converge & Verify

Running converge or verify will create a new instance the first time it is run. The same instance is used for each additional converge or verify.

The test instance policy changed, but no resource explicitly removed or uninstalled the resources defined in the install recipe.

©2016 Chef Software Inc. 1-61 


This is important feature and limitation of using Test Kitchen's 'converge' and 'verify'. Both of these commands will create a test instance the first time they are executed. Every time after these commands will use the same test instance again and again.

When we remove resources from a recipe we do not explicitly uninstall them from the test instance. We simply do not enforce their policy anymore. On an existing system, which this test instance is after the first run, this means it is actually in the desired state that we no longer define. That means that the webserver is still installed, the default web page has still been updated, and the service is still running.

To ensure our cookbook works on a new system it is important to delete the test instance and start over.

Slide 62


CONCEPT



Kitchen Destroy

```
$ kitchen destroy [INSTANCE|REGEXP|all]
```


Destroys one or more instances.



```
graph LR; A[Create CentOS Instance] --> B[Install Chef]; B --> C[Apply the Run List]; C --> D[Execute Tests]; A --> E[Destroyed];
```

©2016 Chef Software Inc.


1-62



Test Kitchen provides the 'destroy' subcommand. Destroy is available at all stages and essentially cleans up the instance. This is useful when you make changes to the configuration policy you define and you want to ensure that it will work on a brand new instance.

Slide 63

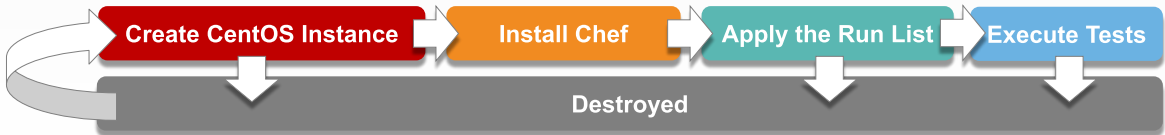
CONCEPT




Kitchen Test

```
$ kitchen test [INSTANCE|REGEXP|all]
```

Destroys (for clean-up), creates, converges, verifies and then destroys one or more instances.




```
graph LR; A[Create CentOS Instance] --> B[Install Chef]; B --> C[Apply the Run List]; C --> D[Execute Tests]; D --> A;
```

©2016 Chef Software Inc. 1-63 

Test Kitchen also provides the subcommand 'test'. Test provides one command that wraps up all the stages in one command. It will destroy any test instance that exists at the start, create a new one, converge the run list on that instance, and then verify it. If everything passes the 'test' subcommand will finish by destroying that instance. If it fails at one of these steps it usually leaves the instance running to allow you to troubleshoot it.

Slide 64


CONCEPT



Kitchen Test

Destroying the instance ensures that the policy is being applied to a new instance.

The test instance is re-created and then the updated policy is applied to the new instance. The new policy is incomplete causing an error.

©2016 Chef Software Inc. 1-64 

Running 'kitchen test' is useful if want to ensure the policy you defined works on a new instance.

Test the Cookbook Against a New Instance




```
> kitchen test
```

```
----> Starting Kitchen (v1.4.2)
----> Cleaning up any prior instances of <default-centos-67>
----> Destroying <default-centos-67>...
...
----> Testing <default-centos-67>
----> Creating <default-centos-67>...
----> Running serverspec test suite
...
Finished in 0.19434 seconds (files took 0.57409 seconds t...
1 example, 1 failure
```

Running 'kitchen test' in this instance will expose the issue that we created by removing that installation of the webserver. This is because the new instance no longer installed the necessary packages so the file path was never created for the default HTML file and there are no services to run.

The test that you wrote correctly verifies the state of the system. What is important to notice is that there are important differences in the Test Kitchen commands.

Slide 66

Converge & Verify	Test
Faster execution time	Slower execution time
Running converge twice will ensure your policy applies without error to existing instances	Running test will ensure your policy applies without error to any new instances
©2016 Chef Software Inc.	1-66
	

Using Test Kitchen to run 'kitchen converge' and 'kitchen verify' is much faster because you are essentially applying and verifying the policy that you have defined against an already running instance. The drawback is that only running 'converge' and 'verify' will not demonstrate for you how your policy will act on a brand new instance.

Using Test Kitchen to run 'kitchen test' is slower because every time you are recreating the test instance, installing chef, and applying the policy on that new instance. The drawback here is the longer feedback cycle and only running 'test' will not demonstrate for you how your policy will act on an existing instance.


Agenda

- ✓ What is Test Driven Development (TDD)?
- ✓ Writing a test first
- ✓ Refactoring with tests
- ❑ Faster Feedback with Unit Tests
- ❑ Further Resources

We refactored the tests and now want to focus on the feedback cycle we have created.


Slide 68

PROBLEM



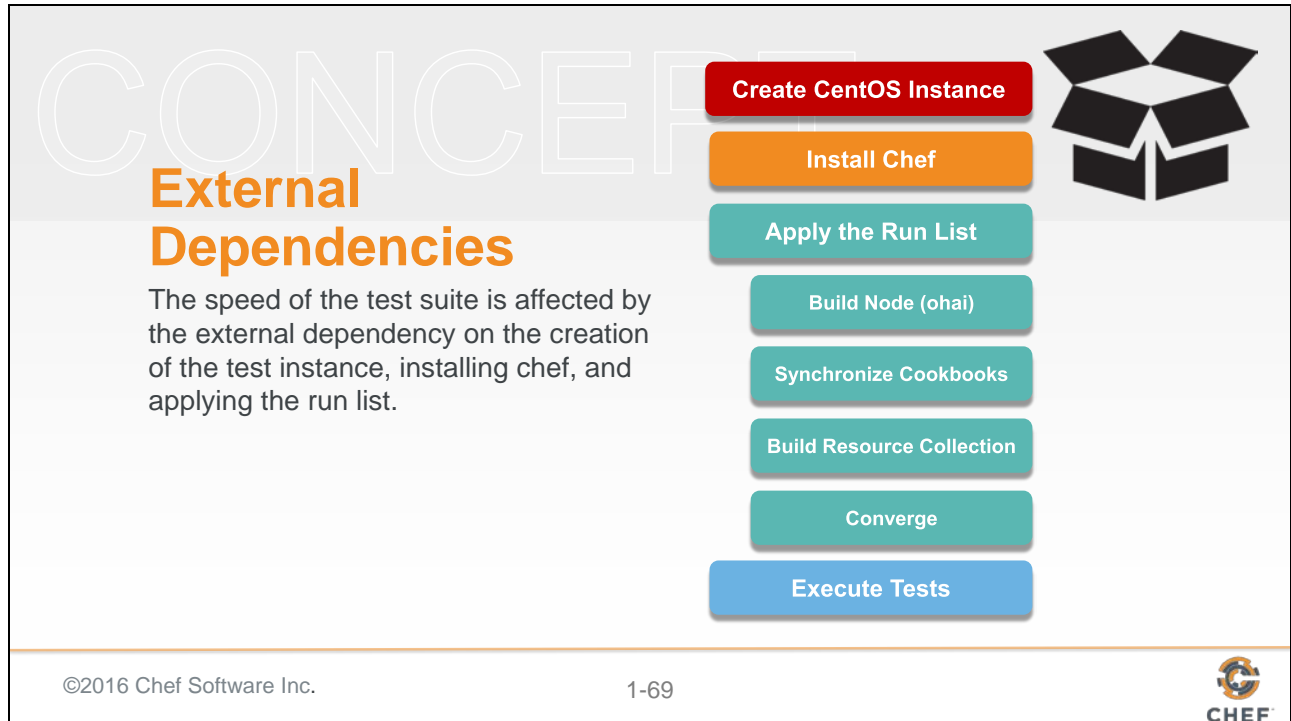
Slower Feedback Cycle

The slower the feedback loop the less value it provides to you while developing your cookbooks. You are less inclined to run the test suite. Which means you will likely miss issues as they happen.

©2016 Chef Software Inc. 1-68 

Interruptions are not conducive to helping you building a flow. To help reduce the interruptive nature of testing we can look at ways to decrease the amount of time you have to wait to receive the feedback from the tests. A faster feedback cycle will increase your likelihood of seeking that feedback again for smaller sets of changes. Slower feedback cycles will increase your likelihood of seeking feedback less often. Causing you create larger sets of changes which has the chance of masking potential issues.

Slide 69



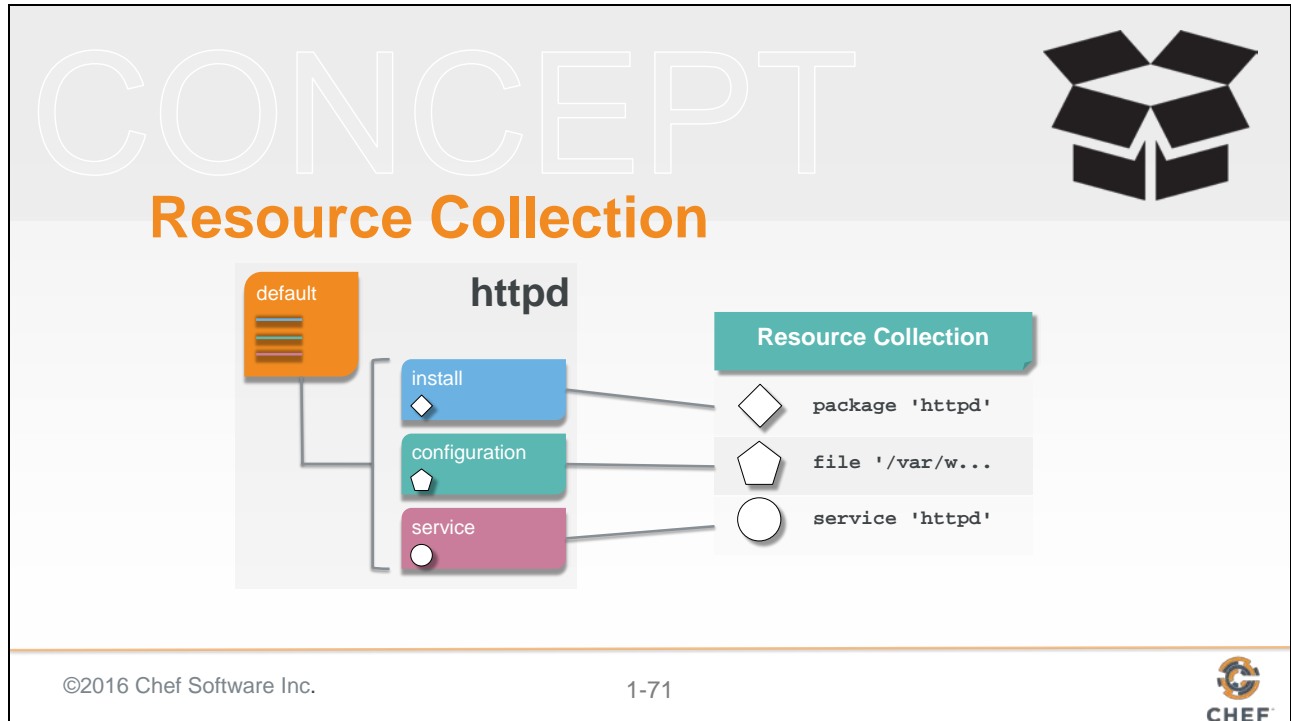
The reason that the feedback cycle takes as long as it does with Test Kitchen is because of the external requirements. Creating the test instance, installing chef, and then applying the run list provide real value because we are able to see the recipe being applied to a virtual instance. However, all these external dependencies incur a time cost as we wait for the network to download images or packages, the test instance's processor to calculate keys or data, or the file-system to create files and folders.

Slide 70



When we mutated our code and executed the test suite we created issues with the resources that we defined and recipes that we included. These changes affected the resources that were applied to the system by omitting resources from the 'Resource Collection'. If we were able to remove the external dependencies and focus on the state of the Resource Collection we would be able to determine if there were problems with the recipes we wrote without the need of any of those external dependencies.

Slide 71




But first let's talk more about the 'Resource Collection' ...

After a cookbook and its recipes have been synchronized the majority of the cookbook content is loaded into memory by 'chef-client'. The recipes defined on the run list are evaluated during this time and the resources found within the recipes and any included recipes, are added to a resource collection. They are not immediately executed like one might assume.

The 'Resource Collection' is almost like a to-do list for the node. It contains the list of all the resources, in order, that need to be accomplished to bring the instance into the desired state. Later, in the converge step, the resources defined in the Resource Collection are executed and perform their various forms of test-and-repair to bring the instance into the desired state.

Slide 72

CONCEPT



RSpec and ChefSpec

RSpec is a Domain Specific Language (DSL) that allows you to express and execute expectations. These expectations are expressed in examples that are asserted in different example groups.

ChefSpec provides helpers and tools that allow you to express expectations about the state of **resource collection**.

ChefSpec


RSpec

Chef

Ruby

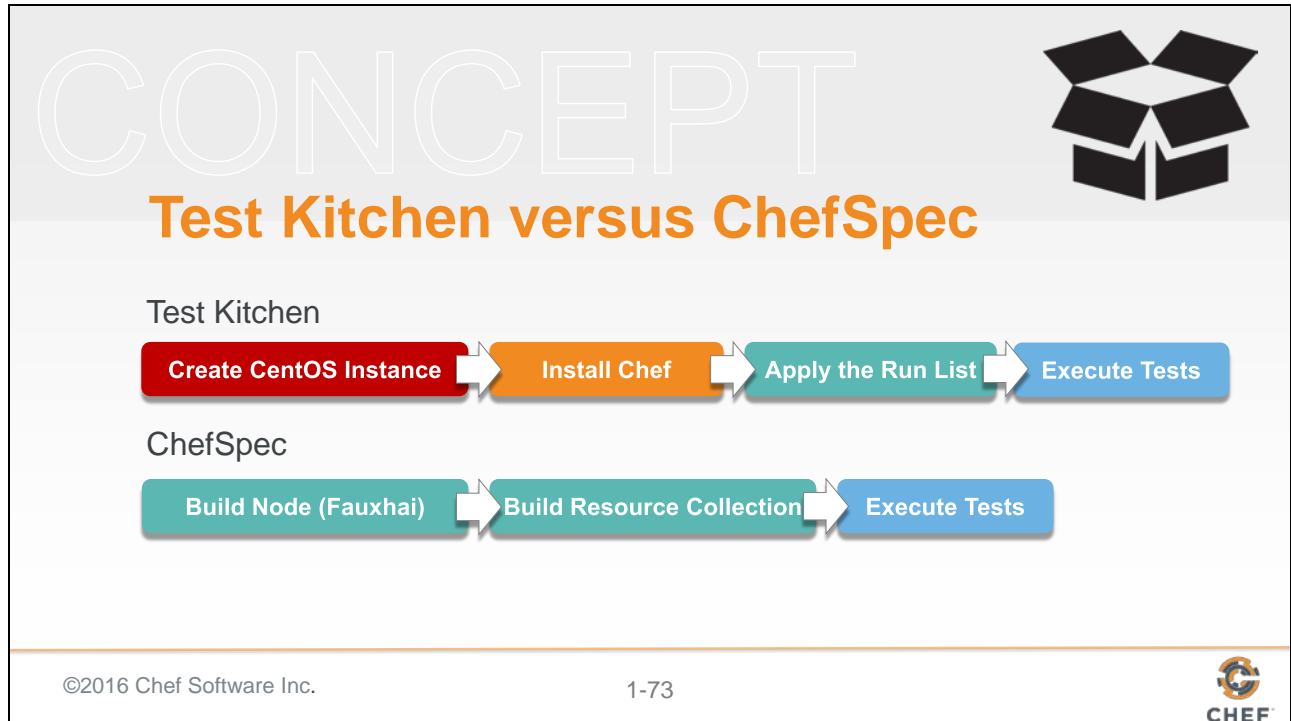
©2016 Chef Software Inc.

1-72



ChefSpec provides a method for us to create an in-memory execution of applying the run list, building the resource collection, and then setting up expectations about the state of the resource collection. ChefSpec, similar to ServerSpec is built on top of RSpec; relying on it to provide the core framework and language. The benefit to us is that a lot of the same language constructs are employed.

Slide 73



Verifying the resource collection with ChefSpec requires far fewer external dependencies and that allows us to get feedback faster but at the cost of not applying the recipes we write against a test instance. This opens us up to situations where we could compose recipes and execute examples that are shown to work because they were correctly added to the resource collection but fail when it comes time for the recipes to apply the desired state against a test instance.

Slide 74

MOTIVATION



To Be Continued ...

Given more time I would have loved to work through some ChefSpec examples.

©2016 Chef Software Inc.

1-74



But unfortunately we don't have time to get to the implementation.

Agenda

- ✓ What is Test Driven Development (TDD)?
- ✓ Writing a test first
- ✓ Refactoring with tests
- ✓ Faster Feedback with Unit Tests
- ❑ Further Resources

Test-Driven Cookbook Development

All the slides and the training content can be found on GitHub:

https://github.com/chef-training/test_driven_cookbook_development

Slide 77

Webinars and Training Videos

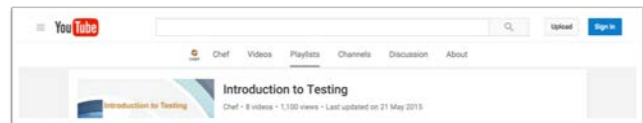
Learn from more more training:

Static Analysis - <https://www.chef.io/webinars/?commid=175731>

Test Kitchen - <https://www.chef.io/webinars/?commid=167803>

Visit and Subscribe to the Chef YouTube channel.

<https://www.youtube.com/user/getchef/playlists>



REFERENCE



Testing Language Documentation

RSpec Documentation (<https://relishapp.com/rspec>)

ChefSpec Documentation (<https://github.com/sethvargo/chefspec>)

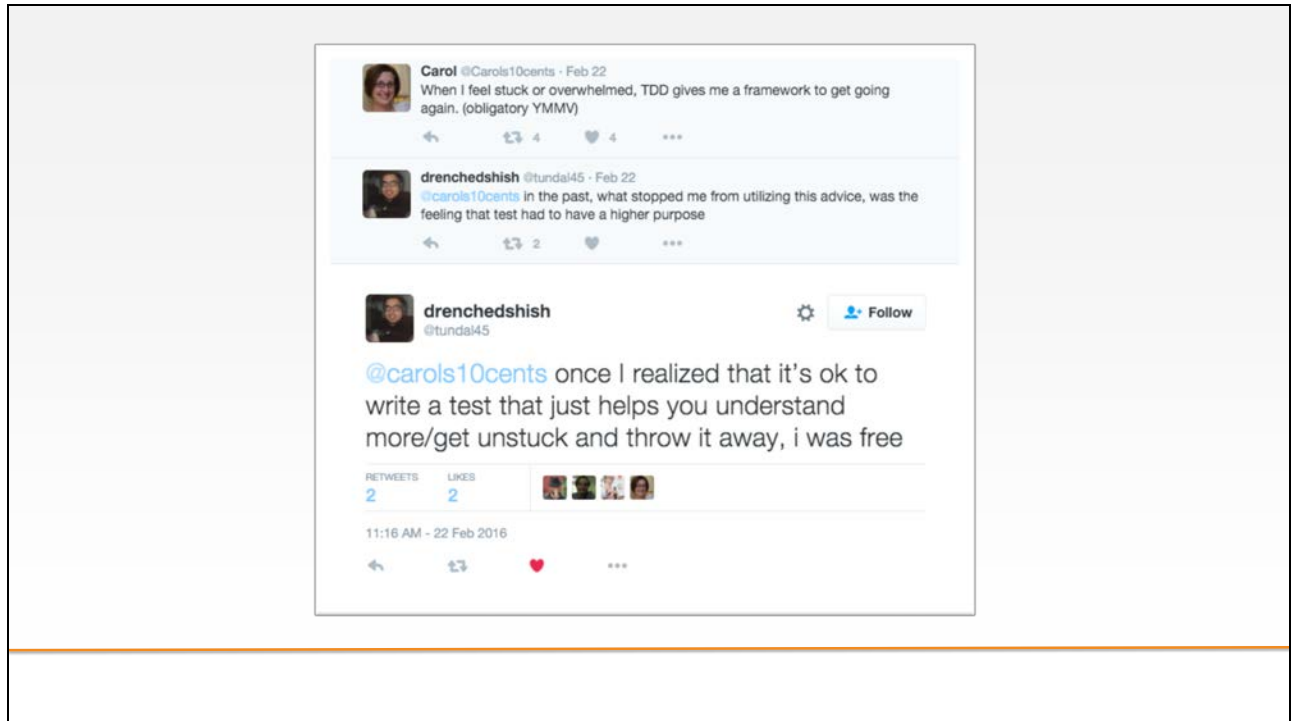
InSpec Documentation (https://docs.chef.io/inspec_reference.html)

ServerSpec Documentation (http://serverspec.org/resource_types.html)

Agenda

- ✓ What is Test Driven Development (TDD)?
- ✓ Writing a test first
- ✓ Refactoring with tests
- ✓ Faster Feedback with Unit Tests
- ✓ Further Resources

Slide 80



Remember writing tests is an exercise to help you understand the product you are building. They are there to help you think. They are only useful if they are helping you.

Slide 81

